Transparent Process Migration for Personal Workstations*

Fred Douglis
John Ousterhout

Computer Science Division
Electrical Engineering and Computer Sciences
University of California
Berkeley, CA 94720
douglis@sprite.Berkeley.EDU

November 17, 1989

Abstract

The Sprite operating system allows executing processes to be moved between hosts at any time. We use this *process migration* mechanism to offload work onto idle machines, and also to *evict* migrated processes when idle workstations are reclaimed by their owners. Sprite's migration mechanism provides a high degree of transparency both for migrated processes and for users. Idle machines are identified, and eviction is invoked, automatically by daemon processes. On Sprite it takes 40–50 milliseconds on DECstation 3100 workstations to perform a remote *exec*. The *pmake* program uses *exec*-time migration to invoke compilations and other tasks concurrently, commonly producing speed-up factors in the range of three to six. Process migration has been in regular service for over a year.

1 Introduction

In a network of personal workstations, many machines are typically idle at any given time. These idle hosts represent a substantial pool of processing power, many times greater than what is available on any user's personal machine in isolation. In recent years a number of mechanisms have been proposed or implemented to harness idle processors (*e.g.*, [14,17, 19,20]). We have implemented process migration in the Sprite operating system for this purpose; this paper is a description of our implementation and our experiences using it.

By "process migration" we mean the ability to move a process's execution site at any time from a *source* machine to a *destination* (or *target*) machine of the same architecture. In practice, process migration in Sprite usually occurs at two particular times. Most often, migration happens as part of the *exec* system call when a resource-intensive program is about to be initiated. *Exec*-time migration is particularly convenient because the process's virtual memory is reinitialized by the *exec* system call and thus need not be transferred from

*This work was supported in part by the Defense Advanced Research Projects Agency under contract N00039-85-C-0269 and in part by the National Science Foundation under grant ECS-8351961.

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE **17 NOV 1989**	2. REPORT TYPE	3. DATES COVERED **00-00-1989 to 00-00-1989**

4. TITLE AND SUBTITLE
Transparent Process Migration for Personal Workstations

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
University of California at Berkeley,Department of Electrical Engineering and Computer Sciences,Berkeley,CA,94720

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT
The Sprite operating system allows executing processes to be moved between hosts at any time. We use this process migration mechanism to offload work onto idle machines, and also to evict migrated processes when idle workstations are reclaimed by their owners. Sprite's migration mechanism provides a high degree of transparency both for migrated processes and for users. Idle machines are identified, and eviction is invoked, automatically by daemon processes. On Sprite it takes 40-50 milliseconds on DECstation 3100 workstations to perform a remote exec. The pmake program uses exec-time migration to invoke compilations and other tasks concurrently, commonly producing speed-up factors in the range of three to six. Process migration has been in regular service for over a year.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**	**Same as Report (SAR)**	**29**	

the source to the target machine. The second common occurrence of migration is when a user returns to a workstation when processes have been migrated to it. At that time all the foreign processes are automatically *evicted* back to their home nodes to minimize their impact on the returning user's interactive response.

Sprite's process migration mechanism provides an unusual degree of *transparency*. Process migration is almost completely invisible both to processes and to users. In Sprite, transparency is defined relative to the *home node* for a process, which is the machine where the process would have executed if there had been no migration at all. A *remote process* (one that has been migrated to a machine other than its home) has exactly the same access to virtual memory, files, devices, and nearly all other system resources that it would have if it were executing on its home node. Furthermore, the process appears to users as if it were still executing on its home machine: its process identifier does not change, it appears in process listings on the home machine, and it may be stopped, restarted, and killed just like other processes. The only obvious sign that a process has migrated is that the load on the source machine suddenly drops and the load on the destination machine suddenly increases.

Although many experimental process migration mechanisms have been implemented, Sprite's is one of only a few to receive extensive practical use (other notable examples are LOCUS [16] and MOSIX [4]). Sprite's migration facility has been in regular use for over a year. Our version of the *make* utility uses process migration automatically so that compilations of different files (and other activities controlled by *make*) are performed concurrently. The speed-up from migration depends on the number of idle machines and the amount of parallelism in the task to be performed, but we commonly see speed-up factors of two or three in compilations and we occasionally obtain speed-ups as high as six or seven. In our environment, about 20-40% of all user activity is performed by processes that are not executing on their home node.

In designing Sprite's migration mechanism, many alternatives were available to us. Our choice among those alternatives consisted of a tradeoff among four factors: transparency, residual dependencies, performance, and complexity. A high degree of transparency implies that processes and users need not act differently after migration occurs than before. If a migration mechanism leaves *residual dependencies*, the source machine must continue to provide some services for a process even after the process has migrated away from it. Residual dependencies are generally undesirable, since they impact the performance of the source machine and make the process vulnerable to failures of the source. By performance, we mean that the act of migration should be efficient and that remote processes should (ideally) execute with the same efficiency as if they hadn't migrated. Lastly, complexity is an important factor because process migration tends to affect virtually every major piece of an operating system kernel. If the migration mechanism is to be maintainable, it is important to limit this impact as much as possible.

Unfortunately, these four factors are in conflict with each other. For example, highly-transparent migration mechanisms are likely to be more complicated and cause residual dependencies. High-performance migration mechanisms may transfer processes quickly at the cost of residual dependencies that degrade the performance of remote processes. A practical implementation of migration must make tradeoffs among the factors to fit the needs of its particular environment. As will be seen in the sections below, we emphasized transparency and performance, but accepted residual dependencies in some situations. (See [2] for another discussion of the tradeoffs in migration, with a somewhat different result.)

A broad spectrum of alternatives also exists for the policy decisions that determine what, when, and where to migrate. For Sprite we chose a semi-automatic approach. The system helps to identify idle hosts, but it does not automatically migrate processes except for eviction. Instead, a few application programs like *pmake* identify long-running processes (perhaps with user assistance) and arrange for them to be migrated to idle machines. When users return to their machines, a system program automatically evicts any processes that had been migrated onto those machines.

The rest of this paper is organized as follows. Section 2 describes the Sprite environment and how it affected our goals for migration. Section 3 discusses why we chose to implement process migration rather than some other simpler mechanism such as remote invocation. Section 4 describes the mechanics of process migration with an emphasis on the two most costly parts of migration in Sprite: virtual memory transfer and open file transfer. In Section 5 we show how Sprite maintains the illusion that a remote process is still executing on its home machine; this is achieved in part by making Sprite kernel calls location-independent, and in part by forwarding operations between the remote machine and the home machine. Section 6 considers the issue of residual dependencies, and Section 7 discusses the policy issues associated with migration. Section 8 reports on the performance of process migration in Sprite, as well as the usage patterns we have experienced thus far. Section 9 discusses our experiences implementing and using migration, and Section 10 concludes the paper.

2 The Sprite Environment

Sprite is an operating system for a collection of personal workstations and file servers on a local area network [15]. Sprite's kernel-call interface is much like that of 4.3 BSD UNIX, but Sprite's implementation is a new one that provides a high degree of network integration. For example, all the hosts on the network share a common high-performance file system. Processes may access files or devices on any host, and Sprite allows file data to be cached around the network while guaranteeing the consistency of shared access to files [13]. Each host runs a distinct copy of the Sprite kernel, but the kernels work closely together using a remote-procedure-call (RPC) mechanism similar to that described by Birrell and Nelson [5].

Four aspects of our environment were particularly important in the design of Sprite's process migration facility:

Idle hosts are plentiful. Since our environment consists primarily of personal machines, it seemed likely to us that many machines would be idle at any given time. For example, Theimer reported that one-third of all machines were typically idle in a similar environment [19]; Nichols reported that 50-70 workstations were typically idle during the day in an environment with 350 workstations total [14]; and our own measurements in Section 8 show 65–85% of all workstations idle on average. The availability of many idle machines suggests that simple algorithms can be used for selecting where to migrate: there is no need to make complex choices among partially-loaded machines.

Users "own" their workstations. A user who is sitting in front of a workstation expects to receive the full resources of that workstation. For migration to be accepted by our users, it seemed essential that migrated processes not degrade interactive response. This suggests that a machine should only be used as a target for migration if it is known to be idle, and that foreign processes should be evicted if the user returns before they finish.

Sprite uses kernel calls. Most other implementations of process migration are in message-passing systems where all communication between a process and the rest of the world occurs through message channels. In these systems, many of the transparency aspects of migration can be handled simply by redirecting message communication to follow processes as they migrate. In contrast, Sprite processes are like UNIX processes in that system calls and other forms of interprocess communication are invoked by making protected procedure calls into the kernel. In such a system the solution to the transparency problem is not as obvious; in the worst case, every kernel call might have to be specially coded to handle remote processes differently than local ones.

Sprite already provides network support. We were able to capitalize on existing mechanisms in Sprite to simplify the implementation of process migration. For example, Sprite already provided remote access to files and devices, and it has a single network-wide space of process identifiers; these features and others made it much easier to provide transparency in the migration mechanism. In addition, process migration was able to use the same kernel-to-kernel remote procedure call facility that is used for the network file system and many other purposes. On DECstation 3100 workstations (roughly 13 MIPS) running on a 10 megabits/second ethernet, the minimum round-trip latency of a remote procedure call is about 1.0 milliseconds and the throughput is 450-650 Kbytes/second. Much of the efficiency of our migration mechanism can be attributed to the efficiency of the underlying RPC mechanism.

To summarize our environmental considerations, we wished to offload work to machines whose users are gone, and to do it in a way that would not be noticed by those users when they returned. We also wanted the migration mechanism to work within the existing Sprite kernel structure, which had one potential disadvantage (kernel calls) and several potential advantages (network-transparent facilities and a fast RPC mechanism).

3 Why Migration?

Much simpler mechanisms than migration are already available for invoking operations on other machines. For example, the BSD versions of UNIX provide a shell command *rsh*, which takes as arguments the name of a machine and a command. *Rsh* causes the given command to be executed on the given remote machine [7].

Rsh has the advantages of being simple and readily available, but there were four reasons why we decided to implement a different mechanism: transparency, eviction, performance, and automatic selection. First, a process created by *rsh* does not run in the same environment as the parent process: the current directory may be different, environment variables are not transmitted to the remote process, and in some systems the remote process may not even have access to the same files and devices as the parent process. In addition, the user has no direct access to remote processes created by *rsh*: the processes do not appear in listings of the user's processes and they cannot be manipulated unless the user logs in to the remote machine. We felt that a mechanism with greater transparency than *rsh* would be easier for users to use.

The second problem with *rsh* is that it does not permit eviction. A process started by *rsh* cannot be moved once it has begun execution. If a user returns to a machine with *rsh*-generated processes, then either the user must tolerate degraded response until the foreign processes complete, or the foreign processes must be killed, which causes work to be lost and annoyance to the user who owns the foreign processes. Nichols' *butler* system terminates foreign processes after warning the user and providing the processes with the opportunity to save their state, but Nichols noted that the ability to migrate existing processes would make *butler* "much more pleasant to use" [14]. Another option is to run foreign processes at low priority so that a returning user receives acceptable interactive response, but this would slow down the execution of the foreign processes. It seemed us to that several opportunities for annoyance could be eliminated, both for the user whose jobs are offloaded and for the user whose workstation is borrowed, by evicting foreign processes when the workstation's user returns.

The third problem with *rsh* is performance. *Rsh* uses standard network protocols with no particular kernel support; the overhead of establishing connections, checking access permissions, and establishing an execution environment may result in delays of several seconds. This makes *rsh* impractical for short-lived jobs and limits the speed-ups that can be obtained using it.

The final problem with *rsh* is that it requires the user to pick a suitable destination machine for offloading. In order to make offloading as convenient as possible for users, we decided to provide an automatic mechanism to keep track of idle machines and select destinations for migration.

Of course, an automated selection mechanism could easily have been layered on top of a remote execution facility like *rsh* (Nichols' *butler* is an example of this approach), and additional transparency features could also have been added, such as preserving the current directory. But at the time we began our design, we saw no way to eliminate all of the above problems within an existing framework. For example, eviction is impossible using *rsh*, as is complete transparency (open files cannot be transmitted from one machine to another). Thus we decided to implement a migration mechanism that allows processes to be moved at any time, to make that mechanism as transparent as possible, and to automate the selection of idle nodes. We felt that this combination of features would encourage the use of migration. We also recognized that this combination of features would probably result in a mechanism much more complex than *rsh*. As a result, one of our key criteria in choosing among implementation alternatives was simplicity.

4 Mechanics of Migration

The techniques used to migrate a process depend on the *state* associated with the process being migrated. If there existed such a thing as a stateless process, then migrating such a process would be trivial. In reality processes have large amounts of state, and both the amount and variety of state seem to be increasing as operating systems evolve. The more state, the more complex the migration mechanism is likely to be. Process state typically includes the following:

- **Virtual memory**. In terms of bytes, the greatest amount of state associated with a process is likely to be the memory that it accesses. Thus the time to migrate a process is limited by the speed of transferring virtual memory.
- **Open files**. If the process is manipulating files or devices, then there will be state associated with these open channels, both in the virtual memory of the process and also in the operating system kernel's memory. The state for an open file includes the internal identifier for the file, the current access position, and possibly cached file

blocks. The cached file blocks may represent a substantial amount of storage, in some cases greater than the process's virtual memory.

- **Message channels.** In a message-based operating system such as Mach [1] or V [6], state of this form would exist in place of open files. (In such a system message channels would be used to access files, whereas in Sprite, file-like channels are used for interprocess communication.) The state associated with a message channel includes buffered messages plus information about senders and receivers.
- **Execution state.** This consists of information that the kernel saves and restores during a context switch, such as register values and condition codes.
- **Other kernel state.** Operating systems typically store other data associated with a process, such as the process's identifier, a user identifier, a current working directory, signal masks and handlers, resource usage statistics, references to the process's parent and children, and so on.

For each portion of the state associated with a process, the system must do one of three things during migration: transfer the state, arrange for forwarding, or ignore the state and sacrifice transparency. To transfer a piece of state, it must be extracted from its environment on the source machine, transmitted to the destination machine, and reinstated in the process's new environment on that machine. For state that is private to the process, such as its execution state, state transfer is relatively straightforward. Other state, such as internal kernel state distributed among complex data structures, may be much more difficult to extract and reinstate. An example of "difficult" state in Sprite is information about open files, particularly those being accessed on remote file servers. Lastly, some state may be impossible to transfer. Such state is usually associated with physical devices on the source machine. For example, the frame buffer associated with a display must remain on the machine containing the display; if a process with access to the frame buffer migrates, it will not be possible to transfer the frame buffer.

The second option for each piece of state is to arrange for forwarding. Rather than transfer the state to stay with the process, the system may leave the state where it is and forward operations back and forth between the state and the process. For example, I/O devices cannot be transferred, but the operating system can arrange for output requests to be passed back from the process to the device, and for input data to be forwarded from the device's machine to the process. In the case of message channels, arranging for forwarding might consist of changing sender and receiver addresses so that messages to and from the channel can find their way from and to the process. Ideally, forwarding should be implemented transparently, so that it is not obvious outside the operating system whether the state was transferred or forwarding was arranged.

The third option, sacrificing transparency, is a last resort: if neither state transfer nor forwarding is feasible, then one can ignore the state on the source machine and simply use the corresponding state on the target machine. The only situation in Sprite where neither state transfer nor forwarding seemed reasonable is for memory-mapped I/O devices such as frame buffers, as alluded to above. In our current implementation, we disallow migration for processes using these devices.

In a few rare cases, lack of transparency may be desirable. For example, a process that requests the amount of physical memory available should obtain information about its current host rather than its home node. For Sprite, a few special-purpose kernel calls, such as to read instrumentation counters in the kernel, are also intentionally non-transparent with respect to migration. In general, though, it would be unfortunate if a process behaved differently after migration than before.

The subsections below describe how Sprite deals with the various components of process state during migration. The solution for each component consists of some combination of transferring state and arranging for forwarding.

4.1 Virtual Memory Transfer

Virtual memory transfer is the aspect of migration that has been discussed the most in the literature, perhaps because it is believed to be the limiting factor in the speed of migration [21]. One simple method for transferring virtual memory is to send the process's entire memory image to the target machine at migration time, as in Charlotte [2] and LOCUS [16]. This approach is simple but it has two disadvantages. First, the transfer can take many seconds, during which time the process is *frozen*: it cannot execute on either the source or destination machine. For some processes, particularly those with real-time needs, long freeze times may be unacceptable. The second disadvantage of a monolithic virtual memory transfer is that it may result in wasted work for portions of the virtual memory that are not used by the process after it migrates. The extra work is particularly unfortunate (and costly) if it requires old pages to be read from secondary storage. For these reasons, several other approaches have been used to reduce the overhead of virtual memory transfer; the mechanisms are diagrammed in Figure 1 and described in the paragraphs below.

In the V System, Theimer addressed the issue of long freeze times with a method called *pre-copying* [18,19]. Rather than freezing a process at the beginning of migration, V allows the process to continue executing while its address space is transferred. Some pages could be modified on the source machine after they have been copied to the destination, so V then freezes the process and copies the pages that were modified. Theimer showed that pre-copying reduces freeze times substantially. However, it has the disadvantage of copying some pages twice, which increases the total amount of work to migrate a process. Pre-copying seems most useful in an environment like V where processes have real-time deadlines.

The Accent system uses a *lazy copying* approach to reduce the cost of process migration [20,21]. When a process migrates in Accent, its virtual memory pages are left on the source machine until they are actually referenced on the target machine. Pages are copied to the target when they are referenced for the first time. This approach allows a process to begin execution on the target with minimal freeze time but introduces many short delays later as pages are retrieved from the source machine. Overall, lazy copying reduces the cost of migration because pages that are not used are never copied at all. Zayas found that for typical programs only one-quarter to one-half of a process's allocated memory needed to be transferred. One disadvantage of lazy copying is that it leaves residual dependencies on the source machine: the source must store the unreferenced pages and provide them on demand to the target. In the worst case, a process that migrates several times could leave virtual memory dependencies on any or all of the hosts on which it ever executed.

Sprite's migration facility uses a different form of lazy copying that takes advantage of our existing network services while providing some of the advantages of lazy copying. In Sprite, backing storage for virtual memory is implemented using ordinary files. Since these backing files are stored in the network file system, they are accessible throughout the network. During migration the source machine freezes the process, flushes its dirty pages to backing files, and discards its address space. On the target machine, the process starts

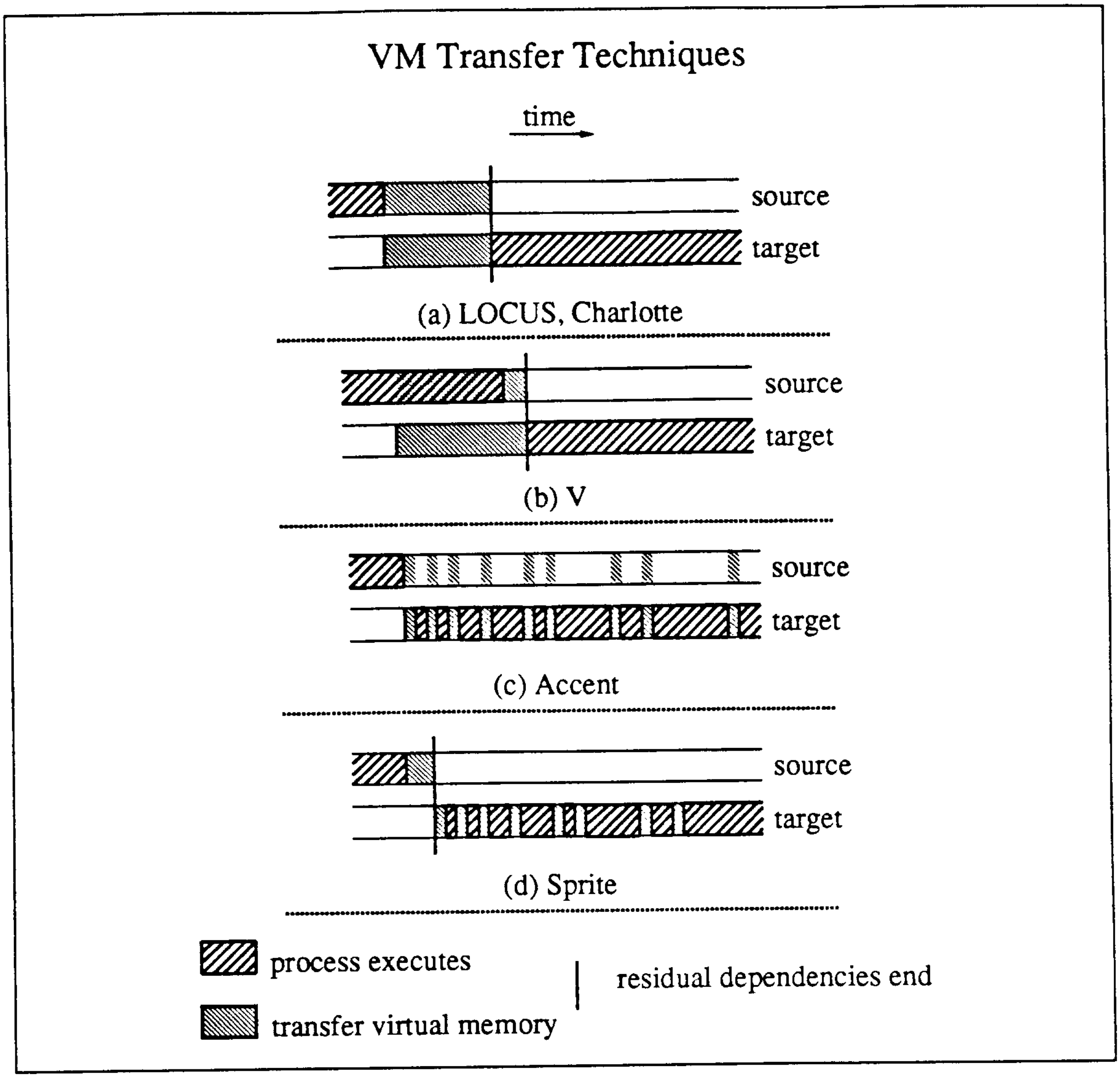

Figure 1: *Different techniques for transferring virtual memory.* (a) shows the scheme used in LOCUS and Charlotte, where the entire address space is copied at the time a process migrates. (b) shows the pre-copying scheme used in V, where the virtual memory is transferred during migration but the process continues to execute during most of the transfer. (c) shows Accent's lazy-copying approach, where pages are retrieved from the source machine as they are referenced on the target. (d) shows Sprite's approach, where dirty pages are flushed to a file server during migration and the target retrieves pages from the file server as they are referenced.

executing with no resident pages and uses the standard paging mechanisms to load pages from backing files as they are needed.

In most cases no disk operations are required to flush dirty pages in Sprite. This is because the backing files are stored on network file servers and the file servers use their memories to cache recently-used file data. When the source machine flushes a dirty page it is simply transferred over the network to the server's main-memory file cache. If the destination machine accesses the page then it is retrieved from the cache. Disk operations will only occur if the server's cache overflows.

Sprite's virtual memory transfer mechanism was simple to implement because it uses pre-existing mechanisms both for flushing dirty pages on the source and for handling page faults on the target. It has some of the benefits of the Accent lazy-copying approach since only dirty pages incur overhead at migration time; other pages are sent to the target machine when they are referenced. Our approach will require more total work than Accent's, though, since dirty pages may be transferred over the network twice: once to a file server during flushing, and once later to the destination machine.

The Sprite approach to virtual memory transfer fits well with the way migration is typically used in Sprite. Process migration occurs most often during an *exec* system call, which completely replaces the process's address space. If migration occurs during an *exec*, the new address space is created on the destination machine so there is no virtual memory to transfer. As others have observed (*e.g.*, LOCUS [16]), the performance of virtual memory transfer for *exec*-time migration is not an issue. Virtual memory transfer *is* an issue, however, when migration is used to evict a process from a machine whose user has returned. In this situation the most important consideration is to remove the process from its source machine quickly, in order to minimize any performance degradation for the returning user. Sprite's approach works well in this regard since (a) it does the least possible work to free up the source's memory, and (b) the source need not retain pages or respond to later paging requests as in Accent. It would have been more efficient overall to transfer the dirty pages directly to the target machine instead of a file server, but this approach would have added complexity to the migration mechanism so we decided against it.

Virtual memory transfer becomes much more complicated if the process to be migrated is sharing writable virtual memory with some other process on the source machine. In principle, it is possible to maintain the shared virtual memory even after one of the sharing processes migrates [11], but this changes the cost of shared accesses so dramatically that it seemed unreasonable to us. Shared writable virtual memory almost never occurs in Sprite right now, so we simply disallow migration for processes using it. A better long-term solution is probably to migrate all the sharing processes together, but even this may be impractical if there are complex patterns of sharing that involve many processes.

4.2 Migrating Open Files

It turned out to be particularly difficult in Sprite to migrate the state associated with open files. This was surprising to us, because Sprite already provided a highly transparent network file system that supports remote access to files and devices; it also allows files to be cached and to be accessed concurrently on different workstations. Thus, we expected that the migration of file-related information would mostly be a matter of reusing existing mechanisms. Unfortunately, process migration introduced new problems in managing the distributed state of open files. Migration also made it possible for a file's current access position to become shared among several machines.

The migration mechanism would have been much simpler if we had chosen the "arrange for forwarding" approach for open files instead of the "transfer state" approach. This would have implied that all file-related kernel calls be forwarded back to the machine where the file was opened, so that the state associated with the file could have stayed on that machine. Because of the frequency of file-related kernel calls and the cost of forwarding a kernel call over the network, we felt that this approach would be unacceptable both because it would slow down the remote process and because it would load the machine that stores the file state. Sprite workstations are typically diskless and files are accessed remotely from file servers, so the forwarding approach would have meant that each file request would be passed over the network once to the machine where the file was opened, and possibly a second time to the server. Instead, we decided to transfer open-file state along with a migrating process and then use the normal mechanisms to access the file (*i.e.*, communicate directly with the file's server).

There are three main components of the state associated with an open file: a file reference, caching information, and an access position. Each of these components introduced problems for migration. The file reference indicates where the file is stored, and also provides a guarantee that the file exists (as required by UNIX semantics): if a file is deleted while open then the deletion is deferred until the file is closed. Our first attempt at migrating files simply closed the file on the source machine and reopened it on the target. Unfortunately, this approach caused files to disappear if they were deleted before the reopen completed. This is such a common occurrence in UNIX programs that file transfer had to be changed to move the reference from source to target without ever closing the file.

The second component of the state of an open file is caching information. Sprite permits the data of a file to be cached in the memory of one or more machines, with file servers responsible for guaranteeing "consistent access" to the cached data [13]. The server for a file keeps track of which hosts have the file open for reading and writing. If a file is open on more than one host and at least one of them is writing it, then caching is disabled: all hosts must forward their read and write requests for that file to the server so they can be serialized. In our second attempt at migrating files, the server was notified of the file's use on the target machine before being told that the file was no longer in use on the source; this made the file appear to be write-shared and caused the server to disable caching for the file unnecessarily. To solve both this problem and the reference problem above we built special server code just for migrating files, so that the transfer from source to destination is made atomically. Migration can still cause caching to be disabled for a file, but only if the file is also in use by some other process on the source machine; if the only use is by the migrating process, then the file will be cacheable on the target machine.

When an open file is transferred during migration, the file cache on the source machine may contain modified blocks for the file. These blocks are flushed to the file's server machine during migration, so that after migration the target machine can retrieve the blocks from the file server without involving the source. This approach is similar to the mechanism for virtual memory transfer and thus has the same advantages and disadvantages. It is also similar to what happens in Sprite for shared file access without migration: if a file is opened, modified, and closed on one machine, then opened on another machine, the modified blocks are flushed from the first machine's cache to the server at the time of the second open.

The third component of the state of an open file is an access position, which indicates where in the file the next read or write operation will occur. Unfortunately the access

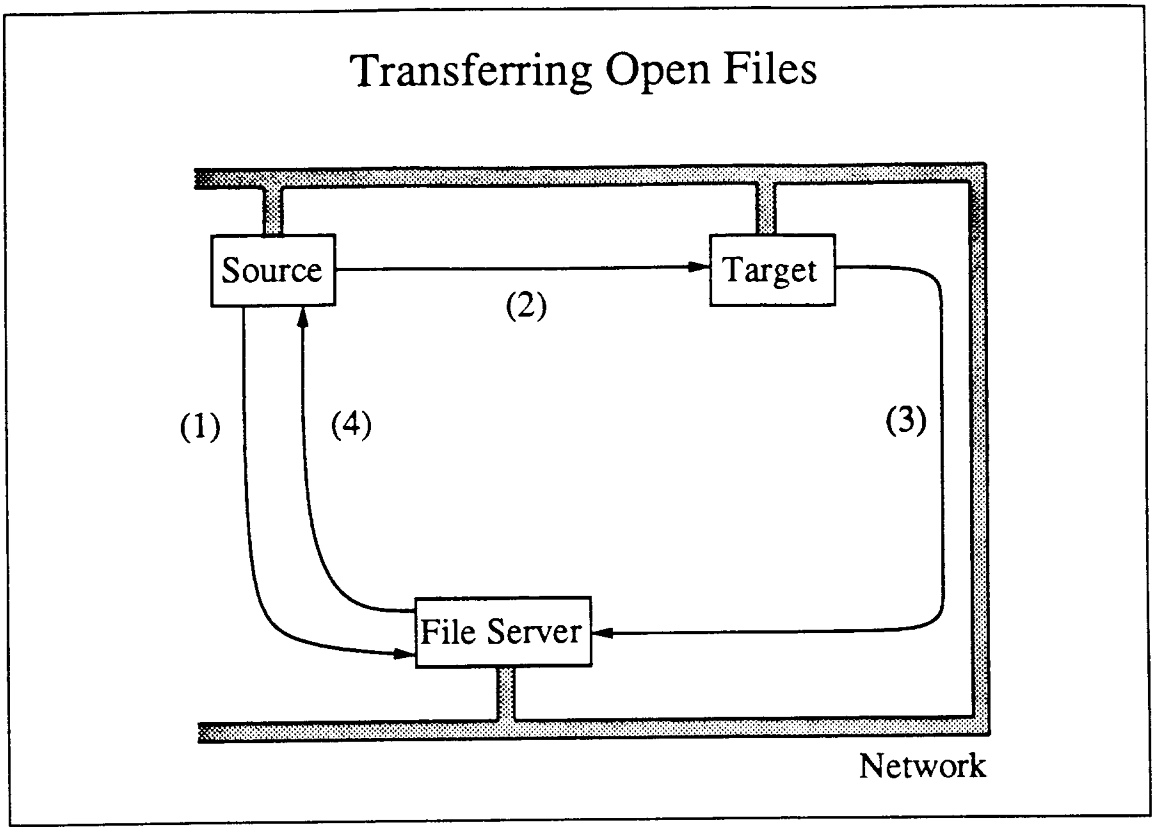

Figure 2: *Transferring open files.* (1) The source notifies the file server that an open file will be migrated, then (2) passes information about the file to the target. (3) The target notifies the server that the open file has been moved; (4) during this call the server communicates again with the source to transfer state about the file.

position for a file may be shared between two or more processes. This happens, for example, when a process opens a file and then forks a child process: the child inherits both the open file and the access position. Under normal circumstances all of the processes sharing a single access position will reside on the same machine, but migration can move one of the processes without the others, so that the access position becomes shared between machines. After several false starts we eventually dealt with this problem in a fashion similar to caching: if an access position becomes shared between machines, then neither machine stores the access position (nor do they cache the file); instead, the file's server maintains the access position and all operations on the file are forwarded to the server.

Another possible approach to shared file offsets is the one used in LOCUS [16]. If process migration causes a file access position to be shared between machines, LOCUS lets the sharing machines take turns managing the access position. In order to perform I/O on a file with a shared access position, a machine must acquire the "access position token" for the file. While a machine has the access position token it caches the access position and no other machine may access the file. The token rotates among machines as needed to give each machine access to the file in turn. This approach is similar to the approach LOCUS uses for managing a shared file, where clients take turns caching the file and pass read and write tokens around to ensure cache consistency.

Figure 2 shows the mechanism currently used by Sprite for migrating open files. The key part of this mechanism occurs in a late phase of migration when the target machine requests that the server update its internal tables to reflect that the file is now in use on the target instead of the source. The server in turn calls the source machine to retrieve information about the file, such as the file's access position and whether the file is in use by other processes on the source machine. This two-level remote procedure call synchronizes the three machines (source, target, and server) and provides a convenient point for updating state about the open file.

4.3 The Process Control Block

Aside from virtual memory and open files, the main remaining issue is how to deal with the process control block (PCB) for the migrating process: should it be left on the source machine or transferred with the migrating process? For Sprite we use a combination of both approaches. The home machine for a process (the one where it would execute if there were no migration) must assist in some operations on the process (see Section 5 for details), so it always maintains a PCB for the process. In addition, the current machine for a process also has a PCB for it. If a process is migrated, then most of the information about the process is kept in the PCB on its current machine; the PCB on the home machine serves primarily to locate the process and most of its fields are unused.

The other elements of process state besides virtual memory and open files are much easier to transfer than virtual memory and open files, since they are not as bulky as virtual memory and they don't involve distributed state like open files. At present the other state consists almost entirely of fields from the process control block. In general, all that needs to be done is to transfer these fields to the target machine and reinstate them in the process control block on the target.

4.4 The Fragility Problem

One problem with process migration is that it involves state that is manipulated by virtually every major module in the kernel. This makes it hard to separate the implementation of migration from the implementation of the other kernel modules, since changes in one can affect the other. As a result, migration has been one of the most fragile parts of the Sprite kernel. It often breaks when seemingly unrelated parts of the kernel are modified. We believe that the problem is inherent in the nature of migration (Theimer had similar problems in his implementation), but we've used two techniques to lessen the difficulties.

Our first approach to the problem of migration fragility was to introduce *migration version numbers*. Before we started using migration version numbers it was very difficult to make any changes in the migration mechanism. During the testing period for the changes, some kernels would be running the new version and some the old. Invariably, attempts would be made to migrate between new and old kernels, and the result was inevitably a crash of one or both machines. The only way to prevent these crashes was to reboot the entire Sprite world whenever we made an incompatible change. We now associate a version number with migration, which is incremented whenever any aspect of the migration protocol changes. Migration is disallowed between machines with different version numbers, so that incompatible versions can coexist in the same network.

Our second approach is to avoid centralizing the migration code in one place. Instead, we have distributed it among the various kernel modules whose state is affected. By placing the migration-related code next to the other code that manipulates state information, we hope it will be easier to keep the two consistent. For each piece of state that must be considered during migration there exist four procedures that are invoked during migration (see Section 4.5 for details). All that is needed to deal with additional state during migration is to write the four procedures for that state and enter their names into a table used during migration. At present the table contains entries for the following pieces of state: basic machine-independent process state, machine-dependent execution state, virtual memory, files, signals, profiling, and arguments to *exec*.

4.5 Migration Procedure

The overall algorithm for migrating a process from a source machine to a target machine is table-driven, as described in the previous section, and involves the following steps:

1. The process is signaled to make it trap into the kernel. (At the point when the signal is handled, the state of the process within the kernel is simple and well-defined.)
2. The target of the migration is contacted to confirm that it is running, that it is available for migration, and that it has the same migration version number as the source. The kernel on the target host allocates a new process control block and returns a token that is later used to identify the new instantiation of the process on the target.
3. For each module with process state that must be transferred to the new host, the source kernel calls a *pre-migration routine* to obtain the size of the encapsulated data. This routine may, as a side effect, initiate any actions required to migrate state: for example, the virtual memory pre-migration routine queues the process's modified pages for flushing.
4. The source kernel allocates a buffer to hold the combined encapsulated state.
5. For each module, the source kernel calls an *encapsulation routine* to place that module's state into a portion of the buffer. As with the pre-migration routines, encapsulation routines may have side effects, such as waiting for modified pages to be flushed or communicating with file servers for open files.
6. The source kernel passes the encapsulated state to the target kernel via a single remote procedure call. On the target, the corresponding *de-encapsulation routine* is invoked for each portion of the buffer.
7. For each module, if a *post-migration routine* has been specified, the kernel on the source host invokes the procedure to clean up any remaining state.
8. The source kernel frees the buffer and informs the target to resume the process.

5 Supporting Transparency: Home Machines

As discussed in Section 3, transparency was one of our most important goals in implementing migration. By "transparency" we mean two things in particular. First, a process's behavior should not be affected by migration. Its execution environment should appear the same, it should have the same access to system resources such as files and devices, and it should produce exactly the same results as if it hadn't migrated. Second, a process's appearance to the rest of the world should not be affected by migration. To the rest of the world the process should appear as if it never left its original machine, and any operation that is possible on an unmigrated process (such as stopping, debugging, or signalling) should be possible on a migrated process.

Both of these two aspects of transparency are defined with respect to a process's *home machine*, which is the machine where it would execute if there were no migration at all. Even after migration, everything should appear as if the process were still executing on its home machine. As will be seen below, Sprite achieves transparency by involving the home machine in some operations for remote processes.

5.1 Messages Versus Kernel Calls

On the surface, it might appear that transparency is particularly easy to achieve in a message-based system like Accent [21], Charlotte [2], or V [6]. In these systems all of a process's interactions with the rest of the world occur in a uniform fashion through message channels. All that is needed to guarantee transparency is to preserve the behavior of the message channels by forwarding messages to and from a remote process's new location. This is typically done by updating addresses in the endpoints of the message channels. For example, Charlotte updates the addresses at the time of migration, while V waits until the old incorrect address is used and then updates the address using a multi-cast protocol.

In contrast, transparency might seem harder to achieve in a system like Sprite that is based on kernel calls. In such a system the state of the process is expected to be in the kernel of the machine where the process executes. This requires that the state be transferred during migration, which is more complicated than forwarding.

It turns out that neither of these initial impressions is correct. For example, it would be possible to implement forwarding in a kernel-call-based system by leaving *all* of the kernel state on the home machine and using remote procedure call to forward home every kernel call [12]. This would result in an approach very similar to forwarding messages, and our initial plan was to use an approach like this for Sprite.

Unfortunately, an approach based entirely on forwarding kernel calls or forwarding messages will not work in practice, for two reasons. The first problem is that some services must necessarily be provided on the machine where a process is executing. If a process invokes a kernel call to allocate virtual memory (or if it sends a message to a memory server to allocate virtual memory), the request *must* be processed by the kernel or server on the machine where the process executes, since only that kernel or server has control over the machine's page tables. Forwarding is not a viable option for such machine-specific functions: state for these operations must be migrated with processes. The second problem with forwarding is cost. It will often be much more expensive to forward an operation to some other machine than to process it locally. If a service is available locally on a remote process's new machine, it will be more efficient to use the local service than to forward operations back to the service on the process's old machine.

5.2 Achieving Transparency

Transparency is achieved in practice by combining several approaches. We used four different techniques in Sprite. The most desirable approach is to make kernel calls location-independent; Sprite has been gradually evolving in this direction. For example, in the early versions of the system we permitted different machines to have different views of the file system name space. This required *open* and several other kernel calls to be forwarded home after migration, imposing about a 20% penalty on the performance of remote compilations. In order to simplify migration (and for several other good reasons also), we changed the file system so that every machine in the network sees the same name space. This made the *open* kernel call location-independent, so no extra effort was necessary to make *open* work transparently for remote processes.

Another example of the evolution toward transparency is the *kill* kernel call, which sends a signal to a process. Sprite has a single network-wide name space for process identifiers, and we made *kill* work regardless of whether the process being signalled was on the same machine as the process issuing the kernel call; this allowed *kill* to be used transparently by remote processes.

Our second technique was to transfer state from the source machine to the target at migration time, so that normal kernel calls may be used after migration. For example, a process's virtual memory is transferred at migration time so that the kernel's virtual-memory-related state for a remote process is identical to that for an unmigrated process. This allowed us to use the normal virtual-memory-related kernel calls for remote processes without any loss of transparency. We also used the state-transfer approach for open files, process and user identifiers, resource usage statistics, and a variety of other things.

Our third technique was to forward kernel calls home. This technique was originally used for a large number of kernel calls, but we have gradually replaced most uses of forwarding with transparency or state transfer. At present there are only a few kernel calls that cannot be implemented transparently and for which we cannot easily transfer state. The most important such kernel call is *gettimeofday*, which returns the current time. Clocks are not synchronized between Sprite machines, so for remote processes Sprite forwards the *gettimeofday* kernel call back to the home machine. This guarantees that time advances monotonically even for remote processes, but incurs a performance penalty for processes that read the time frequently.

Forwarding also occurs from the home machine to a remote process's current machine. For example, when a process is signalled (*e.g.*, when some other process specifies its identifier in the *kill* kernel call), the signal operation is sent initially to the process's home machine. If the process is not executing on the home machine, then the home machine forwards the operation on to the process's current machine. The performance of such operations could be optimized by retaining a cache on each machine of recently used process identifiers and their last known execution sites. This approach is used in LOCUS and V and allows many operations to be sent directly to a remote process without passing through another host. An incorrect execution site is detected the next time it is used and correct information is found by sending a message to the host on which the process was created (LOCUS) or by multi-casting (V).

The fourth "approach" is really just a set of *ad hoc* techniques for a few kernel calls that must update state on both a process's current execution site and its home machine. One example of such a kernel call is *fork*, which creates a new process. Process identifiers in Sprite consist of a home machine identifier and an index of a process within that machine. Management of process identifiers, including allocation and deallocation, is the responsibility of the home machines named in the identifiers. If a remote process *fork*s, the child process must have the same home machine as the parent, which requires that the home machine allocate the new process identifier. Furthermore, the home machine must initialize its own copy of the process control block for the process, as described in Section 4.3. Thus, even though the child process will execute remotely on the same machine as its parent, both its current machine and its home machine must update state. Similar kinds of cooperation occur for *exit*, which is invoked by a process to terminate itself, and *wait*, which is used by a parent to wait for one of its children to terminate. There are several potential race conditions between a process exiting, its parent waiting for it to exit, and one or both processes migrating; we found it easier to synchronize these operations by keeping all the state for the *wait-exit* rendezvous on a single machine (the home). LOCUS similarly uses the site on which a process is created to synchronize operations on the process.

6 Residual Dependencies

We define a *residual dependency* as an on-going need for a host to maintain data structures or provide functionality for a process even after the process migrates away from the host. One example of a residual dependency occurs in Accent, where a process's virtual memory pages are left on the source machine until they are referenced on the target. Another example occurs in Sprite, where the home machine must participate whenever a remote process forks or exits.

Residual dependencies are undesirable for three reasons: reliability, performance, and complexity. Residual dependencies decrease reliability by allowing the failure of one host to affect processes on other hosts. Residual dependencies decrease performance for the remote process because they require remote operations where local ones would otherwise have sufficed. Residual dependencies also add to the load of the host that is depended upon, thereby reducing the performance of other processes executing on that host. Lastly, residual dependencies complicate the system by distributing a process's state around the network instead of concentrating it on a single host; a particularly bad scenario is one where a process can migrate several times, leaving residual dependencies on every host it has visited.

Despite the disadvantages of residual dependencies, it may be impractical to eliminate them all. In some cases dependencies are inherent, such as when a process is using a device on a specific host; these dependencies cannot be eliminated without changing the behavior of the process. In other cases, dependencies are necessary or convenient to maintain transparency. Lastly, residual dependencies may actually improve performance in some cases, such as lazy copying in Accent, by deferring state transfer until it is absolutely necessary.

In Sprite we were much more concerned about transparency than about reliability, so we permitted some residual dependencies on the home machine where those dependencies made it easier to implement transparency. As described in Section 5 above, there are only a few situations where the home machine must participate so the performance impact is minimal (see Section 8 for measurements).

Although Sprite permits residual dependencies on the home machine, it does not leave dependencies on any other machines. If a process migrates to a machine and is then evicted or migrates away for any other reason, there will be no residual dependencies on that machine. This provides yet another assurance that process migration will not impact users' response when they return to their workstations. The only noticeable long-term effect of foreign processes is the resources they may have utilized during their execution: in particular, the user's virtual memory working set may have to be demand-paged back into memory upon the user's return.

The greatest drawback of residual dependencies on the home node is the inability of users to migrate processes in order to survive the failure of their home node. We are considering a nontransparent variant of process migration, which would change the home node of a process when it migrates and break all dependencies on its previous host.

7 Migration Policies

Until now we have focussed our discussion on the *mechanisms* for transferring processes and supporting remote execution. This section considers the *policies* that determine how migration is used. Migration policy decisions fall into four categories:

What. Which processes should be migrated? Should all processes be considered candidates for migration, or only a few particularly CPU-intensive processes? How are CPU-intensive processes to be identified?

When. Should processes only be migrated at the time they are initiated, or may processes also be migrated after they have been running?

Where. What criteria should be used to select the machines that will be the targets of migration?

Who. Who makes all of the above decisions? How much should be decided by the user and how much should be automated in system software?

At one end of the policy spectrum lies the *pool of processors* model. In this model the processors of the system are treated as a shared pool and all of the above decisions are made automatically by system software. Users submit jobs to the system without any idea of where they will execute. The system assigns jobs to processors dynamically, and if process migration is available it may move processes during execution to balance the loads of the processors in the pool. MOSIX [4] is one example of the "pool of processors" model: processors are shared equally by all processes and the system dynamically balances the load throughout the system, using process migration.

At the other end of the policy spectrum lies *rsh*, which provides no policy support whatsoever. In this model individual users are responsible for locating idle machines, negotiating with other users over the use of those machines, and deciding which processes to offload.

For Sprite we chose an intermediate approach where the selection of idle hosts is fully automated but the other policy decisions are only partially automated. There were two reasons for this decision. First, our environment consists of personal workstations. Users are happy running almost all of their processes locally on their own personal workstations, and they expect to have complete control of their workstations. Users do not think of their workstations as "shared". Second, the dynamic pool-of-processors approach appeared to us to involve considerable additional complexity, and we were not convinced that the benefits would justify the implementation difficulties. For example, most processes in a UNIX-like environment are so short-lived that migration will not produce a noticeable benefit and may even slow things down. Eager et al. provide additional evidence that migration is only useful under particular conditions [10]. Thus, for Sprite we decided to make migration a special case rather than the normal case.

The Sprite kernels provide no particular support for any of the migration policy decisions, but user-level applications provide assistance in four forms: idle-host selection, the *pmake* program, a *mig* shell command, and eviction. These are discussed in the following subsections.

7.1 Selecting Idle Hosts

Each Sprite machine runs a background process called the *load-average daemon*, which monitors the usage of that machine. When the workstation appears to be idle, the load-average daemon notifies the rest of the system that the machine is ready to accept migrated processes. Programs that invoke migration, such as *pmake* and *mig* described below, call a standard library procedure *Mig_GetIdleNode* to return the identifier of an idle host, which they then pass to the kernel when they invoke migration. No other process is allowed to use that host for remote execution until the process relinquishes it via the *Mig_Done* library call. (In some environments, multiple processes might be able to make effective use of a single idle host, but the applications that have used the migration facility thus far work well with exclusive access.)

Maintaining the database of idle hosts can be a challenging problem in a distributed system, particularly if the system is very large in size or if there are no shared facilities available for storing load information. A number of distributed algorithms have been proposed to solve this problem, such as disseminating load information among hosts periodically [4], querying other hosts at random to find an idle one [9], or multicasting and accepting a response from any host that indicates availability [18].

Since Sprite has a shared network file system, we were able to store the idle-host database in a shared file. The load-average daemons set flags in the file when their hosts become idle, and the *Mig_GetIdleNode* library procedure selects an idle host at random from the file, marking the selected host so that no one else selects it. Standard file-locking primitives are used to synchronize access to the file.

The shared-file approach is a particularly simple one, but it has a number of disadvantages. It does not retain state from request to request, making it difficult to implement features like reassigning the same machine to the same user (in order to reuse files that might have been cached on the machine), or fair allocation of idle hosts when there are more would-be users than idle machines. Although these features could conceivably be implemented with a shared file, there would be a high overhead from repeated communication with a file server. The shared-file approach also provides inadequate protection of information about idle hosts, since the shared file must be made readable and writable by all users. It requires explicit deallocation of hosts when they are no longer used by an application, and the abnormal termination of a program using idle hosts can leave a host marked as in-use for a prolonged period of time. We are currently exploring a different approach where a centralized server process maintains the database of idle hosts, makes allocation decisions, and detects when hosts are no longer in use for remote execution.

We chose a conservative set of criteria for determining whether a machine is "idle". The load-average daemon currently will only consider a host to be idle if (a) it has had no keyboard or mouse input for at least five minutes, and (b) there are fewer runnable processes than processors, on average. In choosing these criteria we wanted to be certain not to inconvenience active users or delay background processes they might have left running. We assumed that there would usually be plenty of idle machines to go around, so we were less concerned about using them efficiently. In Section 10 below, we discuss our choice of criteria in light of our experience.

7.2 Pmake and Mig

Sprite provides two convenient ways to use migration. The most common use of process migration is by the *pmake* program. *Pmake* is similar in function to the *make* UNIX utility and is used, for example, to detect when source files have changed and recompile the corresponding object files. *Make* performs its compilations and other actions serially; in contrast, *pmake* uses process migration to invoke as many commands in parallel as there are idle hosts available. This use of process migration is completely transparent to users and results in substantial speed-ups in many situations (see Section 8 for performance measurements). Other systems besides Sprite have also benefitted from parallel make facilities; see [3] and [17] for examples.

The approach used by *pmake* has at least one advantage over a fully-automatic "processor pool" approach where all the migration decisions are made centrally. Because *pmake* makes the choice of processes to offload, and knows how many hosts are available, it can scale its parallelism to match the number of idle hosts. If the offloading choice were made by some other agent, *pmake* might overload the system by creating more processes than could be accommodated efficiently. *Pmake* also provides a degree of flexibility by permitting the user to specify that certain tasks should not be offloaded if they are poorly suited for remote execution.

The second easy way to use migration is with a program called *mig*, which takes as argument a shell command. *Mig* will select an idle node using the mechanism described above and use process migration to execute the given command on that machine. *Mig* may also be used to migrate an existing process.

Although these are the only two mechanisms commonly used for process migration at present, we hope that our users will find other ways to use migration to speed up their applications.

7.3 Eviction

The final form of system support for migration is eviction. The load-average daemons described in Section 7.1 detect when a user returns. On the first keystroke or mouse-motion invoked by the user, the load-average daemon will check for foreign processes and evict them. At present, eviction causes the processes to be migrated back to their home machines. A better approach would be to check first for other idle machines and migrate the foreign processes there rather than home, but automatic remigration has not been a high priority thus far because remote applications are relatively short-lived.

8 Performance and Usage Patterns

We evaluated process migration in Sprite by taking three sets of measurements. Section 8.1 discusses particular operations in isolation, such as the time to migrate a trivial process or invoke a remote command. Section 8.2 describes the performance improvement of *pmake* using parallel remote execution. Section 8.3 presents empirical measurements of Sprite's process migration facility over a period of several weeks, including the extent to

Action	Time/Rate
Select & release idle host	56 milliseconds
Migrate "null" process	43 milliseconds
Transfer info for open files	6.1 milliseconds/file
Flush modified file blocks	450 Kbytes/second
Flush modified pages	640 Kbytes/second
Fork, *exec* null process with migration, wait for child to exit	41 milliseconds
Fork, *exec* null process locally, wait for child to exit	21 milliseconds

Table 1: *Costs associated with process migration.* All measurements were performed on DECstation 3100 workstations. Host selection may be amortized across several migrations if applications such as *pmake* reuse idle hosts. The time to migrate a process depends on how many open files the process has and how many modified blocks for those files are cached locally (these must be flushed to the server). If the migration is not done at *exec*-time, modified virtual memory pages must be flushed as well. The *execs* were performed with no open files.

which migration is used, the cost and frequency of eviction, and the availability of idle hosts. Section 8.4 draws some conclusions based on these measurements.

8.1 Migration Overhead

Table 1 summarizes the costs associated with migration. Host selection on DECstation 3100 workstations takes an average of 56 milliseconds, which includes numerous file system interactions to obtain and then release the idle host. Process transfer is a function of some fixed overhead, plus variable overhead in proportion to the number of modified virtual memory pages and file blocks copied over the network and the number of files the process has open. If a process *execs* at the time of migration, as is normally the case, no virtual memory is transferred.

All things considered, it takes about a tenth of a second to select an idle host and start a new process on it, not counting any time needed to transfer open files or flush modified file blocks to servers. This latency may be too great to warrant running trivial programs remotely, but it is substantially less than the time needed to compile typical source programs, run text formatters, or do any number of other CPU-bound tasks.

After a process migrates away from its home node, it may suffer from the overhead of forwarding system calls. The degradation due to remote execution depends on the ratio of location-dependent system calls to other operations, such as computation and file I/O. Figure 3 shows the total execution time to run several programs, listed in Table 2, both entirely locally and entirely on a single remote host. Applications that communicate frequently with the home node suffered considerable degradation. Two of the benchmarks, *fork* and *gettime*, are contrived examples of the type of degradation a process might experience if it performed many location-dependent system calls without much user-level computation. The *rcp* benchmark is a more realistic example of the penalties processes can encounter: it copies data using TCP, and TCP operations are sent to a user-level TCP server on the home node. Forwarding these TCP operations causes *rcp* to perform about 25% more slowly

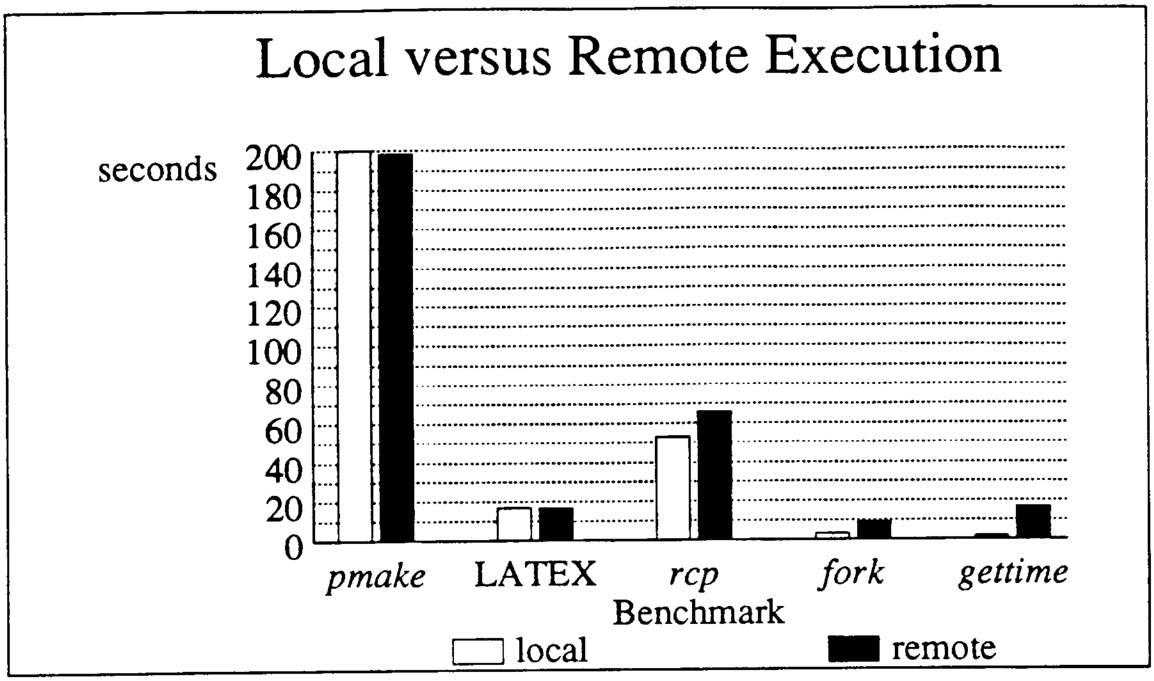

Figure 3: *Comparison between local and remote execution of programs.* CPU-intensive applications such as *pmake* and LaTeX showed negligible effects from remote execution. Other applications suffered performance penalties ranging from 24% (*rcp*), to 260% (*fork*), to 3200% (*gettime*).

Name	Description
pmake	recompile *pmake* source sequentially using *pmake*
LaTeX	run LaTeX on a draft of this paper
rcp	copy a 1 Mbyte file to another host using TCP
fork	fork and wait for child, 1000 times
gettime	get the time of day 10000 times

Table 2: *Workload for comparisons between local and remote execution.*

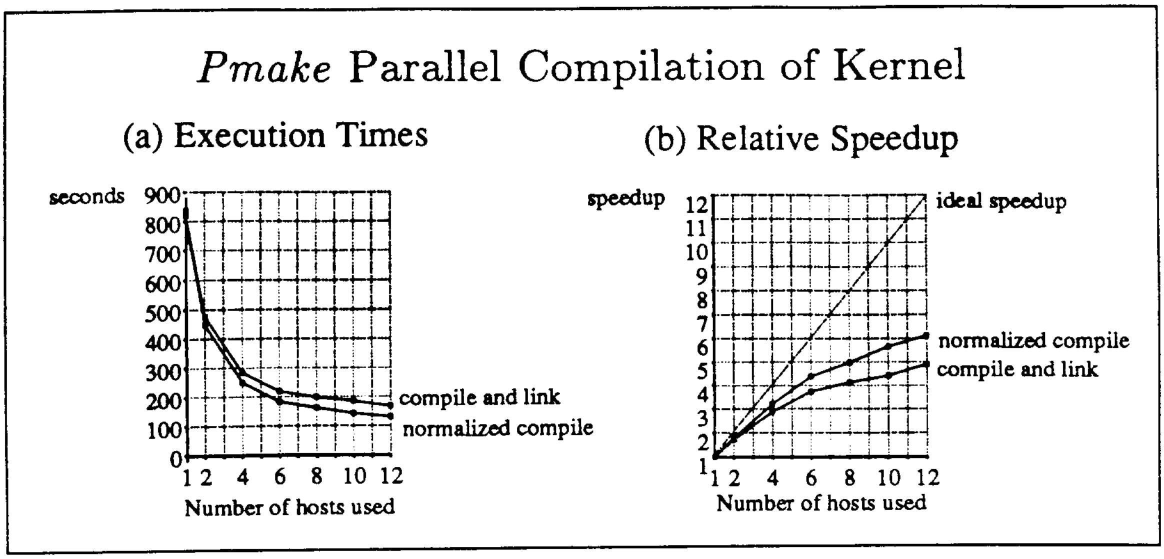

Figure 4: *Performance of recompiling the Sprite kernel using a varying number of hosts and the* pmake *program.* Graph (a) shows the time to compile all the C files and then link the resulting object files into a single file. In addition, it shows a "normalized" curve that shows the time taken just to compile, further adjusted by the *pmake* startup overhead of 8 seconds to determine dependencies; this curve represents the parallelizable portion of the *pmake* benchmark. Graph (b) shows the speed-up obtained for each point in (a), which is the ratio between the time taken on a single host and the time using multiple hosts in parallel.

when run remotely than locally. As may be seen in Figure 3, however, applications such as compilations and text formatting show little effect from remote execution. (In fact, executing *pmake* was slightly faster remotely than locally, due to obscure differences in process scheduling while performing remote procedure calls.)

8.2 Applications Performance

The benchmarks in Section 8.1 measure the component costs of migration. This section measures the overall benefits of migration using *pmake*. The first benchmark consists of compiling 139 Sprite kernel source files, then linking the resulting object files into a single file. Each *pmake* command (compiling or linking) is performed on a remote host using *exec*-time migration. Once a host is obtained from the pool of available hosts, it is reused until *pmake* finishes or the host is no longer available.

Figure 4 shows the total elapsed time to compile and link the Sprite kernel using a varying number of machines in parallel, as well as the performance improvement obtained. In this environment, *pmake* is able to make effective use of about three-fourths of each host it uses up to a point (4-6 hosts), but it uses only half the processing power available to it once additional hosts are used.

The "compile and link" curve in Figure 4(b) shows a speed-up factor of 5 using 12 hosts. Clearly, there is a significant difference between the speed-ups obtained for the "normalized compile" benchmark and the "compile and link" benchmark. The difference is

Program	Number of		Sequential Time	Parallel Time	Speed-Up
	Files	Links			
gremlin	24	1	202	44	4.59
TEX	36	1	416	80	5.20
pmake	49	3	200	60	3.33
kernel	139	1	818	167	4.90

Table 3: *Examples of* pmake *performance.* Sequential execution is done on a single host; parallel execution uses migration to execute up to 12 tasks in parallel. Each measurement gives the time to compile the indicated number of files and link the resulting object files together in one or more steps. When multiple steps are required, their sequentiality reduces the speed-up that may be obtained; *pmake*, for example, is organized into two directories that are compiled and linked separately, and then the two linked object files are linked together.

partly attributable to the sequential parts of running *pmake*: determining file dependencies and linking object files all must be done on a single host. More importantly, file caching affects speed-up substantially. As described above in Section 4.2, when a host opens a file for which another host is caching modified blocks, the host with the modified blocks transfers them to the server that stores the file. Thus, if *pmake* uses many hosts to compile different files in parallel, and then a single host links the resulting object files together, that host must wait for each of the other hosts to flush the object files they created. It then must obtain the object files from the server. In this case, linking the files together when they have all been created on a single host takes only 8 seconds, but the link step takes 20–35 seconds when multiple hosts are used for the compilations.

In practice, we don't even obtain the five-fold speed-up indicated by this benchmark, because we compile and link each kernel module separately and link the modules together afterwards. Each link step is an additional synchronization point that may be performed by only one host at a time. In our development environment, we typically see three to four times speed-up when rebuilding a kernel from scratch. Table 3 presents some examples of typical *pmake* speed-ups. These times are representative of the performance improvements seen in day-to-day use.

Figure 4 and Table 3 demonstrate that the benefits of using idle hosts in parallel adequately compensate for the combined overhead of host selection, migration, and forwarding location-dependent operations. The number of hosts that may be effectively used depends in part upon the relative speeds of the file server(s) and the hosts performing the compilation. On Sun 3/75 workstations, for example, we often obtain a speed-up nearly equal to the number of hosts used, for small degrees of parallelism (up to four to six hosts); however, with 10 hosts compiling in parallel, the marginal improvement is small and in one case a Sun 3/180 file server's processor was in use 90% of the time. Twelve DECstation 3100 workstations compiling in parallel used only 55% of a DECstation file server, which suggests that faster (or multiprocessor) file servers may alleviate potential server bottlenecks.

Host	Total CPU Time	Remote CPU Time	Fraction Remote
Host1	241000 secs	97775 secs	40.57 %
Host2	40387	582	1.44 %
Host3	38797	6164	15.89 %
Host4	31841	8372	26.29 %
Host5	28592	8228	28.78 %
Host6	18248	4254	23.31 %
Host7	12411	1690	13.62 %
Host8	5847	2320	39.68 %
Host9	5259	0	0.00 %
Host10	3198	1797	56.19 %
Others	4043	1232	30.47 %
Total	429623	132414	30.82 %

Table 4: *Remote processing use over a two-week period.* The ten hosts with the greatest total processor usage are shown individually. Sprite hosts performed roughly 30% of user activity using process migration. The standard deviation of the fraction of remote use was 21%.

8.3 Usage Patterns

We instrumented Sprite to keep track of remote execution, evictions, and the availability of idle hosts. First, when a process exits, the total time during which it executed is added to a global counter; if the process had been executing remotely, its time is added to a separate counter as well. (These counters therefore exclude some long-running processes that do not exit before a host reboots; however, these processes are daemons, display servers, and other processes that would normally be unsuitable for migration.) Since we started recording these statistics, remote processes have accounted for about 30% of all processing done on Sprite. Some hosts run applications that make much greater use of remote execution, executing as much as 50–60% of user cycles on other hosts. Table 4 lists some sample processor usage over this period.

Second, we record each time a host changes from *idle* to *active*, indicating that foreign processes would be evicted if they exist, and we count the number of times evictions actually occur. To date, evictions have been extremely rare. On the average, each host changes to the *active* state only once every couple of hours, and very few of these transitions actually result in processes being evicted (0.01 processes per hour per host so far in a collection of 14 DECstations). When evictions do occur, they take about one second to migrate an average of about 3 processes. An average of 54 4-Kbyte pages are written per process that migrates. Thus, given the infrequency of eviction and the size of their modified memory, the efficiency of transferring the virtual address space is probably not an issue. Once the remote execution facility is used for larger applications, such as simulators, the average modified address space size is likely to increase.

Third, over the course of a couple of months, we periodically recorded the state of every host (active, idle, or hosting foreign processes) in a log file. A surprisingly large number (60–80%) of hosts are available for migration at any time, even during the day on

Time Frame	In Use	Idle	In Use for Migration
weekdays	30 %	65 %	5 %
off-hours	7 %	90 %	3 %
total	13 %	84 %	3 %

Table 5: *Host availability.* Weekdays are Monday through Friday from 9:00 A.M. to 5:00 P.M. Off-hours are all other times.

weekdays. This is partly due to our environment, in which several users own both a Sun and a DECstation and use only one or the other at a time. Some workstations are available for public use and are not used on a regular basis. However, after discounting for extra workstations, we still find a sizable fraction of hosts available, concurring with Theimer, Nichols, and others. Table 5 summarizes the availability of hosts in Sprite over this period.

To further study the availability of idle hosts, we logged requests for idle hosts over a three-month period. Each process that used the migration library wrote a log entry containing the number of requests for idle hosts it made and the number of requests that could be satisfied. During this period, over 30,000 processes requested one or more idle hosts, and 76% of those processes obtained as many hosts as they requested. Somewhat surprisingly, given the wide availability of idle hosts, 13% of the processes were unable to obtain any hosts at all. Since *pmake* tries to obtain as many hosts as it has tasks to perform, it is possible for one *pmake* to acquire every idle host of the same architecture and prevent other processes from using idle hosts. The centralized server mentioned above in Section 7.1 will be able to enforce fairness constraints and prevent the starvation we have experienced with a simple shared file.

8.4 Observations

Based on our experience, as well as those of others (V [18], Charlotte [2], and Accent [21]), we have observed the following:

- The overall improvement from using idle hosts can be substantial, depending upon the degree of parallelism in an application.
- Remote execution current accounts for a sizable fraction of all processing on Sprite. Even so, idle hosts are plentiful. Our use of idle hosts is currently limited more by a lack of applications (other than *pmake*) than by a lack of hosts.
- The cost of *exec*-time migration is high by comparison to the cost of local process creation, but it is relatively small compared to times that are noticeable by humans. Furthermore, the overhead of providing transparent remote execution in Sprite is negligible for most classes of processes. The system may therefore be liberal about placing processes on other hosts at *exec* time, as long as the likelihood of eviction is relatively low.
- The cost of transferring a process's address space and flushing modified file blocks dominates the cost of migrating long-running processes, thereby limiting the effectiveness of a dynamic "pool of processors" approach. Although there are other environments

in which such an approach could have many favorable aspects, given the assumptions in Section 2 about host availability and workstation "ownership", using process migration to balance the load among all Sprite hosts would likely be both unnecessary and undesirable.

9 History and Experience

The greatest lesson we have learned from our experience with process migration is the old adage "use it or lose it." Although an experimental version of migration was operational in 1986 [8], it took another two years to make migration a useful utility. Part of the problem was that a few important mechanisms weren't implemented initially (*e.g.*, there was no automatic host selection, migration was not integrated with *pmake*, and process migration did not deal gracefully with machine crashes). But the main problem was that migration continually broke due to other changes in the Sprite kernel. Without regular use, problems with migration weren't noticed and tended to accumulate. As a result, migration was only used for occasional experiments. Before each experiment a major effort was required to fix the accumulated problems, and migration quickly broke again after the experiment was finished.

By the fall of 1988 we were beginning to suspect that migration was too fragile to be maintainable. Before abandoning it we decided to make one last push to make process migration completely usable, integrate it with the *pmake* program, and use it for long enough to understand its benefits as well as its drawbacks. This was a fortunate decision. Within one week after migration became available in *pmake*, other members of the Sprite project were happily using it and achieving speed-up factors of two to five in compilations. Because of its complex interactions with the rest of the kernel, migration is still more fragile than we would like and it occasionally breaks in response to other changes in the kernel. However, it is used so frequently that problems are detected immediately and they can usually be fixed quickly. Today we consider migration to be an indispensable part of the Sprite system.

We are not the only ones to have had difficulties keeping process migration running: for example, Theimer reported similar experiences with his implementation in V [18]. Part of the problem is inherent in migration, since it interacts with many other parts of the kernel. However, we've taken two steps to improve the maintainability of migration. The first was to distribute the migration code among other kernel modules as described in Section 4.4. The second approach was to sensitize the kernel developers to the presence of migration, so that they consider the potential impact on migration when making other changes. The maintenance load is still higher for migration than for many other parts of the kernel, but only slightly.

The most complicated aspects of migration are those related to migrating open files. In particular, locking and updating the data structures for an open file on multiple hosts provided numerous opportunities for distributed deadlocks, race conditions, and inconsistent reference counts. Process migration has the unpleasant property of turning shared state into *distributed* shared state, which is much more difficult to manage. The access position of a file is one example of this effect (see Section 4.2).

We are often asked whether it is really necessary to migrate processes at arbitrary times during their execution. In our current usage patterns virtually all migration occurs at *exec-*

time except for occasional evictions; if migration were permitted only at *exec*-time, wouldn't the implementation be simpler and more maintainable? Furthermore, the processes we migrate now tend to be short-lived ones like compilations. Is eviction really necessary? Why not either kill the foreign processes when a user returns (since not much work would be lost), or just let them complete (which would take only a few seconds on average)?

Exec-only migration would definitely be simpler than our current version since several pieces of state would not need to be migrated, such as virtual memory and much of the process's execution state. Unfortunately, open files would still need to be transferred since they are inherited across *exec* calls, and many of the issues about maintaining transparency during remote execution would remain. Thus much of the implementation complexity would still be present unless transparency were sacrificed (e.g., by not retaining open files across migration). The loss of eviction might not make much difference in our current environment, but it would make migration painful to use for long-running tasks such as simulations: the tasks would have to check-point their state so that migrated processes could be killed and restarted elsewhere when users return to their machines. In order to encourage the greatest possible use of migration we plan to retain our current mechanism, which allows migration anytime but uses it most often at *exec*-time.

10 Conclusions

Process migration is now taken for granted as an essential part of the Sprite system. It is used hundreds of times daily and provides substantial speed-ups for applications that are amenable to coarse-grain parallel processing, such as compilation. The transparency provided by the migration mechanism makes it easy to use migration, and eviction keeps migration from bothering the people whose machines are borrowed. As more people use Sprite, we are discovering new applications for process migration, such as long-running simulations. Collectively, remote execution accounts for a sizable portion of all user activity on Sprite.

To date, we have intentionally been conservative in our use of migration in order to gain acceptance among our users. We do not use a workstation unless it has been idle long enough that its owner is unlikely to use it again soon, and only *pmake* commands are typically executed remotely by default. However, our measurements suggest that evictions would be infrequent, and inexpensive, even if we were more liberal about placing remote processes. In addition to encouraging the development of parallel simulators and other user applications for migration, we intend to explore new system-wide migration tools: for example, a shell (command executor) that invokes background processes on idle hosts when possible. We also plan to reduce the constraints on idle time to increase host availability along with the demand for idle hosts; we may then assess any changes in the frequency and cost of evictions and remote execution.

From the outset we expected migration to be difficult to build and maintain. Even so, we were surprised at the complexity of the interactions between process migration and the rest of the kernel, particularly where distributed state was involved as with open files. It was interesting that Sprite's network file system both simplified migration (by providing transparent remote access to files and devices) and complicated it (because of the file system's complex distributed state). We believe that our implementation has now reached a stable and maintainable state, but it has taken us a long time to get there.

For us, the bottom line is that process migration is too useful to pass up. We encourage others to make process migration available in their systems, but to beware of the implementation pitfalls.

11 Acknowledgments

In addition to acting as guinea pigs for the early unstable implementations of process migration, other members of the Sprite project have made significant contributions to Sprite's process migration facility. Mike Nelson and Brent Welch implemented most of the mechanism for migrating open files, and Adam de Boor wrote the *pmake* program. Lisa Bahler, Thorsten von Eicken, John Hartman, Darrell Long, Mendel Rosenblum, and Ken Shirriff provided comments on early drafts of this paper, which improved the presentation substantially.

References

[1] M. J. Accetta *et al.*, "Mach: A new kernel foundation for UNIX development," in *Proceedings of the USENIX 1986 Summer Conference*, July 1986.

[2] Y. Artsy and R. Finkel, "Designing a process migration facility: The Charlotte experience," *IEEE Computer*, vol. 22, pp. 47–56, Sept. 1989.

[3] E. H. Baalbergen, "Parallel and distributed compilations in loosely-coupled systems: A case study," in *Proceedings of Workshop on Large Grain Parallelism*, (Providence, RI), Oct. 1986.

[4] A. Barak, A. Shiloh, and R. Wheeler, "Flood prevention in the MOSIX load-balancing scheme," *IEEE Computer Society Technical Committee on Operating Systems Newsletter*, vol. 3, pp. 23–27, Winter 1989.

[5] A. D. Birrell and B. J. Nelson, "Implementing remote procedure calls," *ACM Transactions on Computer Systems*, vol. 2, pp. 39–59, Feb. 1984.

[6] D. R. Cheriton, "The V distributed system," *Communications of the ACM*, vol. 31, pp. 314–333, Mar. 1988.

[7] Computer Science Division, University of California, Berkeley, *UNIX User's Reference Manual, 4.3 Berkeley Software Distribution, Virtual VAX-11 Version*, Apr. 1986.

[8] F. Douglis and J. Ousterhout, "Process migration in the Sprite operating system," in *Proceedings of the 7th International Conference on Distributed Computing Systems*, (Berlin, West Germany), pp. 18–25, IEEE, Sept. 1987.

[9] D. L. Eager, E. D. Lazowska, and J. Zahorjan, "Adaptive load sharing in homogeneous distributed systems," *IEEE Transactions on Software Engineering*, vol. SE-12, pp. 662–675, May 1986.

[10] D. L. Eager, E. D. Lazowska, and J. Zahorjan, "The limited performance benefits of migrating active processes for load sharing," in *ACM SIGMETRICS 1988*, May 1988.

[11] K. Li and P. Hudak, "Memory coherence in shared virtual memory systems," in *Proceedings of the 5th ACM Symposium on Principles of Distributed Computing*, pp. 229–239, ACM, Aug. 1986.

[12] M. Litzkow, "Remote UNIX," in *Proceedings of the USENIX 1987 Summer Conference*, June 1987.

[13] M. Nelson, B. Welch, and J. Ousterhout, "Caching in the Sprite network file system," *ACM Transactions on Computer Systems*, vol. 6, pp. 134–154, Feb. 1988.

[14] D. Nichols, "Using idle workstations in a shared computing environment," in *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, (Austin, TX), pp. 5–12, ACM, November 1987.

[15] J. Ousterhout *et al.*, "The Sprite network operating system," *IEEE Computer*, vol. 21, pp. 23–36, Feb. 1988.

[16] G. J. Popek and B. J. Walker, eds., *The LOCUS Distributed System Architecture*. Computer Systems Series, The MIT Press, 1985.

[17] E. Roberts and J. Ellis, "*parmake* and *dp*: Experience with a distributed, parallel implementation of make," in *Proceedings from the Second Workshop on Large-Grained Parallelism*, Software Engineering Institute, Carnegie-Mellon University, Nov. 1987. Report CMU/SEI-87-SR-5.

[18] M. Theimer, *Preemptable Remote Execution Facilities for Loosely-Coupled Distributed Systems*. PhD thesis, Stanford University, 1986.

[19] M. Theimer, K. Lantz, and D. Cheriton, "Preemptable remote execution facilities for the V-System," in *Proceedings of the 10th Symposium on Operating System Principles*, pp. 2–12, Dec. 1985.

[20] E. Zayas, "Attacking the process migration bottleneck," in *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, (Austin, TX), pp. 13–22, November 1987.

[21] E. Zayas, *The Use of Copy-On-Reference in a Process Migration System*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, Apr. 1987. Report No. CMU-CS-87-121.